Control-Flow Graph and Local Optimizations - Part 2

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

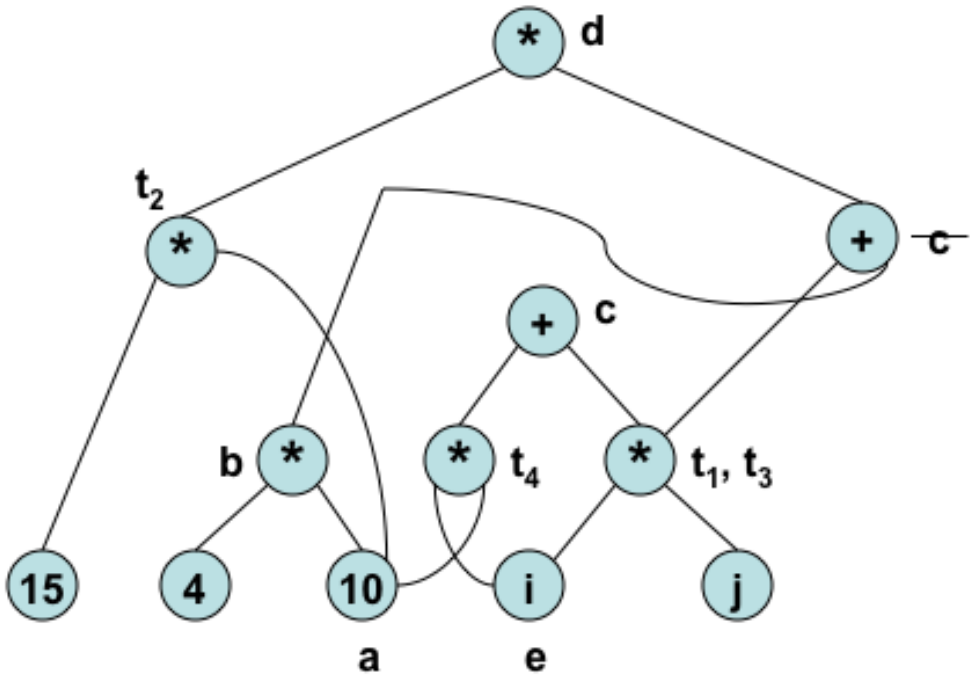
NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- What is code optimization and why is it needed? (in part 1)
- Types of optimizations (in part 1)
- Basic blocks and control flow graphs (in part 1)
- Local optimizations (in part 1)
- Building a control flow graph (in part 1)
- Directed acyclic graphs and value numbering

Example of a Directed Acyclic Graph (DAG)

1. $a = 10$
2. $b = 4 * a$
3. $t1 = i * j$
4. $c = t1 + b$
5. $t2 = 15 * a$
6. $d = t2 * c$
7. $e = i$
8. $t3 = e * j$
9. $t4 = i * a$
10. $c = t3 + t4$



Value Numbering in Basic Blocks

- A simple way to represent DAGs is via *value-numbering*
- While searching DAGs represented using pointers etc., is inefficient, *value-numbering* uses hash tables and hence is very efficient
- Central idea is to assign numbers (called value numbers) to expressions in such a way that two expressions receive the same number if the compiler can prove that they are equal for all possible program inputs
- We assume quadruples with binary or unary operators
- The algorithm uses three tables indexed by appropriate hash values:
  HashTable, ValnumTable, and *NameTable*
- Can be used to eliminate common sub-expressions, do constant folding, and constant propagation in basic blocks
- Can take advantage of commutativity of operators, addition of zero, and multiplication by one

Data Structures for Value Numbering

In the field *Namelist*, first name is the defining occurrence and replaces all other names with the same value number with itself (or its constant value)

HashTable entry
(indexed by expression hash value)

Expression	Value number

ValnumTable entry
(indexed by name hash value)

Name	Value number

NameTable entry
(indexed by value number)

Name list	Constant value	Constflag

Example of Value Numbering

HLL Program	Quadruples before Value-Numbering	Quadruples after Value-Numbering
$a = 10$	1. $a = 10$	1. $a = 10$
$b = 4 * a$	2. $b = 4 * a$	2. $b = 40$
$c = i * j + b$	3. $t1 = i * j$	3. $t1 = i * j$
$d = 15 * a * c$	4. $c = t1 + b$	4. $c = t1 + 40$
$e = i$	5. $t2 = 15 * a$	5. $t2 = 150$
$c = e * j + i * a$	6. $d = t2 * c$	6. $d = 150 * c$
	7. $e = i$	7. $e = i$
	8. $t3 = e * j$	8. $t3 = i * j$
	9. $t4 = i * a$	9. $t4 = i * 10$
	10. $c = t3 + t4$	10. $c = t1 + t4$ (Instructions 5 and 8 can be deleted)

Running the algorithm through the example (1)

1. $a = 10$:
   - a is entered into *ValnumTable* (with a *vn* of 1, say) and into *NameTable* (with a constant value of 10)
2. $b = 4 * a$:
   - a is found in *ValnumTable*, its constant value is 10 in *NameTable*
     - We have performed *constant propagation*
     - $4 * a$ is evaluated to 40, and the quad is rewritten
     - We have now performed *constant folding*
     - b is entered into *ValnumTable* (with a *vn* of 2) and into *NameTable* (with a constant value of 40)
3. $t1 = i * j$:
   - i and j are entered into the two tables with new *vn* (as above), but with no constant value
   - $i * j$ is entered into *HashTable* with a new *vn*
   - $t1$ is entered into *ValnumTable* with the same *vn* as $i * j$

Running the algorithm through the example (2)

4. Similar actions continue till $e = i$
   - e gets the same *vn* as i
5. $t3 = e * j$:
   - e and i have the same *vn*
   - hence, $e * j$ is detected to be the same as $i * j$
   - since $i * j$ is already in the HashTable, we have found a *common subexpression*
   - from now on, all uses of $t3$ can be replaced by $t1$
   - quad $t3 = e * j$ can be deleted
6. $c = t3 + t4$:
   - $t3$ and $t4$ already exist and have *vn*
   - $t3 + t4$ is entered into *HashTable* with a new *vn*
   - this is a reassignment to c
   - c gets a different *vn*, same as that of $t3 + t4$
7. Quads are renumbered after deletions

Example: *HashTable* and *ValNumTable*

HashTable

Expression	Value-Number
$i * j$	5
$t1 + 40$	6
$150 * c$	8
$i * 10$	9
$t1 + t4$	11

ValNumTable

Name	Value-Number
a	1
b	2
i	3
j	4
$t1$	5
c	6,11
$t2$	7
d	8
e	3
$t3$	5
$t4$	10

Handling Commutativity etc.

- When a search for an expression $i + j$ in *HashTable* fails, try for $j + i$
- If there is a quad $x = i + 0$, replace it with $x = i$
- Any quad of the type, $y = j * 1$ can be replaced with $y = j$
- After the above two types of replacements, value numbers of x and y become the same as those of i and j, respectively
- Quads whose LHS variables are used later can be marked as *useful*
- All unmarked quads can be deleted at the end

Handling Array References

Consider the sequence of quads:

1. $X = A[i]$
2. $A[j] = Y$: i and j could be the same
3. $Z = A[i]$: in which case, $A[i]$ is not a common subexpression here

- The above sequence cannot be replaced by: $X = A[i];\ A[j] = Y;\ Z = X$
- When $A[j] = Y$ is processed during value numbering, ALL references to array A so far are searched in the tables and are marked KILLED - this kills quad 1 above
- When processing $Z = A[i]$, killed quads not used for CSE
- Fresh table entries are made for $Z = A[i]$
- However, if we know apriori that $i \neq j$, then $A[i]$ can be used for CSE

Handling Pointer References

Consider the sequence of quads:

1. $X = *p$
2. $*q = Y$: p and q could be pointing to the same object
3. $Z = *p$: in which case, $*p$ is not a common subexpression here

- The above sequence cannot be replaced by: $X = *p;\ *q = Y;\ Z = X$
- Suppose no pointer analysis has been carried out
  - p and q can point to *any* object in the basic block
  - Hence, When $*q = Y$ is processed during value numbering, ALL table entries created so far are marked KILLED - this kills quad 1 above as well
  - When processing $Z = *p$, killed quads not used for CSE
  - Fresh table entries are made for $Z = *p$

Handling Pointer References and Procedure Calls

- However, if we know apriori which objects *p* and *q* point to, then table entries corresponding to only those objects need to killed
- Procedure calls are similar
- With no dataflow analysis, we need to assume that a procedure call can modify any object in the basic block
  - changing call-by-reference parameters and global variables within procedures will affect other variables of the basic block as well
- Hence, while processing a procedure call, ALL table entries created so far are marked KILLED
- Sometimes, this problem is avoided by making a procedure call a separate basic block

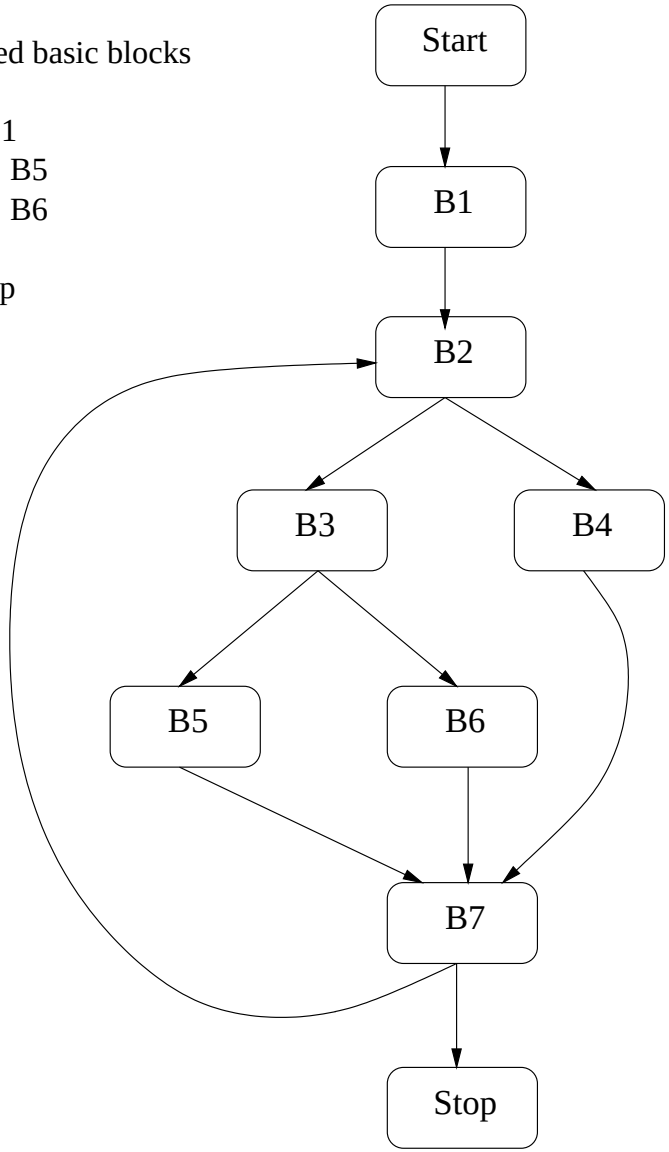
Extended Basic Blocks

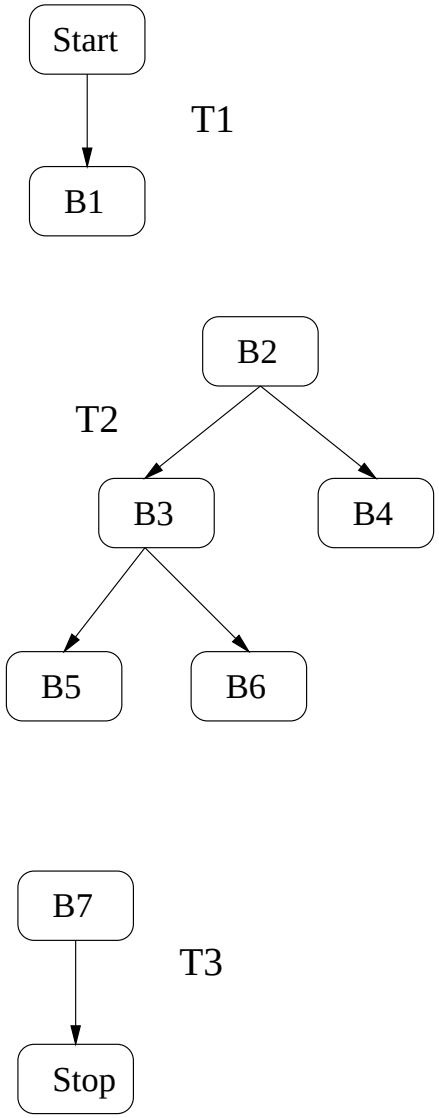
- A sequence of basic blocks $B_1, B_2, ..., B_k$, such that B_i is the unique predecessor of $B_{i+1} (i \leq i < k)$, and B_1 is either the start block or has no unique predecessor
- Extended basic blocks with shared blocks can be represented as a tree
- Shared blocks in extended basic blocks require scoped versions of tables
- The new entries must be purged and changed entries must be replaced by old entries
- Preorder traversal of extended basic block trees is used

Extended Basic Blocks and their Trees

Extended basic blocks

Start, B1
B2, B3, B5
B2, B3, B6
B2, B4
B7, Stop





Value Numbering with Extended Basic Blocks

function $visit\text{-}ebb\text{-}tree(e)$ // e is a node in the tree
begin
  // From now on, the new names will be entered with a new scope into the tables.
  // When searching the tables, we always search beginning with the current scope
  // and move to enclosing scopes. This is similar to the processing involved with
  // symbol tables for lexically scoped languages
  $value\text{-}number(e.B)$;
  // Process the block $e.B$ using the basic block version of the algorithm
  if ($e.left \neq null$) then $visit\text{-}ebb\text{-}tree(e.left)$;
  if ($e.right \neq null$) then $visit\text{-}ebb\text{-}tree(e.right)$;
  remove entries for the new scope from all the tables
  and undo the changes in the tables of enclosing scopes;
end

begin // main calling loop
  for each tree t do $visit\text{-}ebb\text{-}tree(t)$;
  // t is a tree representing an extended basic block
end

Machine Code Generation - 1

Y. N. Srikant
Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- Machine code generation – main issues
- Samples of generated code
- Two Simple code generators
- Optimal code generation
  - Sethi-Ullman algorithm
  - Dynamic programming based algorithm
  - Tree pattern matching based algorithm
- Code generation from DAGs
- Peephole optimizations

Code Generation – Main Issues (1)

- Transformation:
  - Intermediate code → m/c code (binary or assembly)
  - We assume quadruples and CFG to be available
- Which instructions to generate?
  - For the quadruple A = A+1, we may generate
    - Inc A *or*
    - Load A, R1
      Add #1, R1
      Store R1, A
  - One sequence is faster than the other (cost implication)

Code Generation – Main Issues (2)

- In which order?
  - Some orders may use fewer registers and/or may be faster
- Which registers to use?
  - Optimal assignment of registers to variables is difficult to achieve
- Optimize for memory, time or power?
- Is the code generator easily retargetable to other machines?
  - Can the code generator be produced automatically from specifications of the machine?

Samples of Generated Code

- B = A[i]

```
Load   i, R1 // R1 = i
Mult   R1,4,R1// R1 = R1*4
// each element of array
// A is 4 bytes long
Load   A(R1), R2// R2=(A+R1)
Store  R2, B// B = R2
```

- X[j] = Y

```
Load  Y, R1// R1 = Y
Load  j, R2// R2 = j
Mult   R2, 4, R2// R2=R2*4
Store  R1, X(R2)// X(R2)=R1
```

- X = *p

```
Load   p, R1
Load   0(R1), R2
Store  R2, X
```

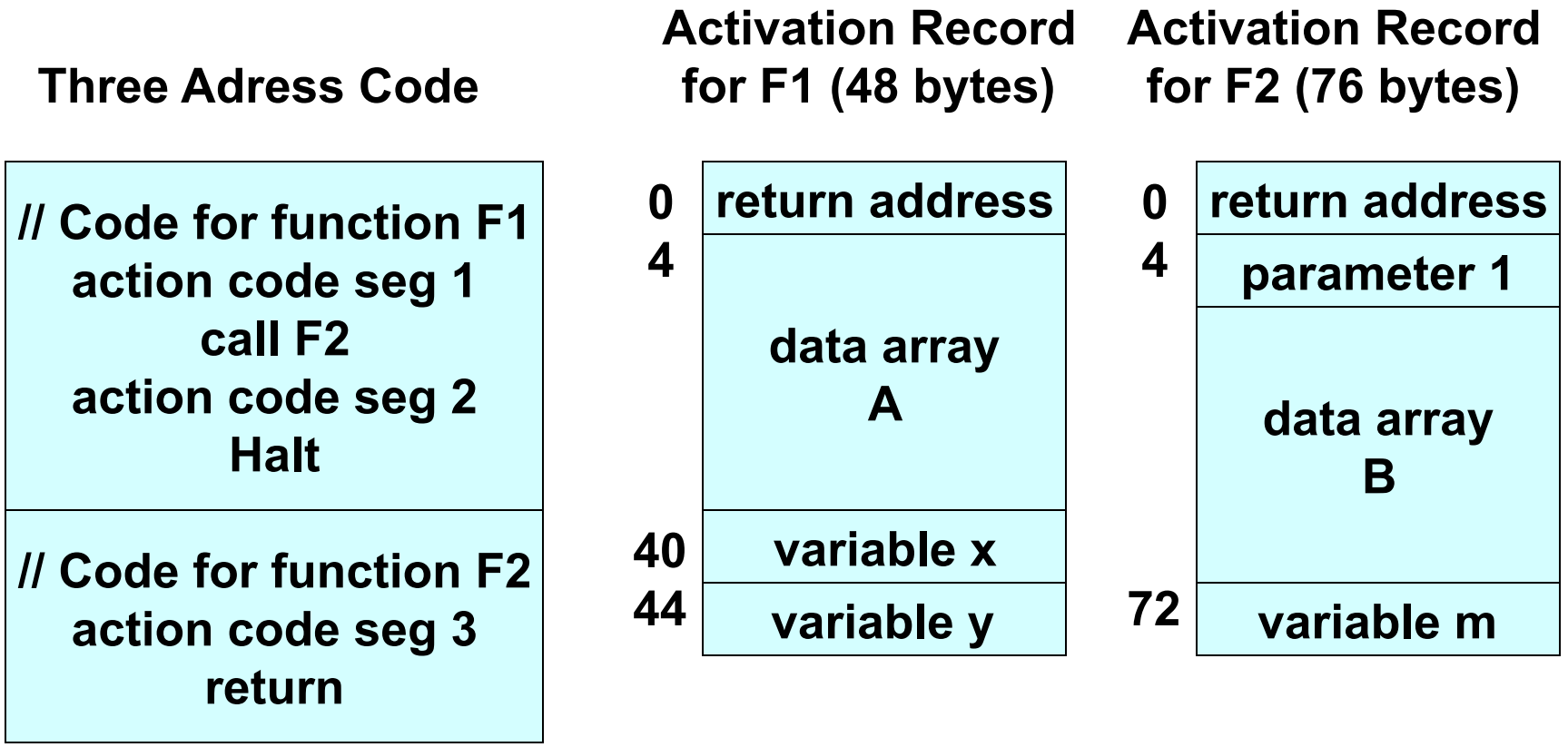
- *q = Y

```
Load   Y, R1
Load   q, R2
Store   R1, 0(R2)
```

- if X < Y goto L

```
Load  X, R1
Load  Y, R2
Cmp   R1, R2
Bltz    L
```

Samples of Generated Code – Static Allocation (no JSR instruction)



Samples of Generated Code – Static Allocation (no JSR instruction)

```
// Code for function F1
200:    Action code seg 1
// Now store return address
240:    Move #264, 648
252:    Move val1, 652
256:    Jump 400 // Call F2
264:    Action code seg 2
280:    Halt
        ...
// Code for function F2
400:    Action code seg 3
// Now return to F1
440:    Jump @648
        ...
```

```
//Activation record for F1
//from 600-647
600:    //return address
604:    //space for array A
640:    //space for variable x
644:    //space for variable y
//Activation record for F2
//from 648-723
648:    //return address
652:    // parameter 1
656:    //space for array B
        ...
720:    //space for variable m
```